Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Emil Itäjärvi

iOS-sovellusten yhdistämisprojekti

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja Viestintätekniikka

Insinöörityö

26.5.2020

Tekijä Otsikko Sivumäärä Aika	Emil Itäjärvi iOS-sovellusten yhdistämisprojekti 40 sivua 26.5.2020
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja Viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander Ohjaaja Mia Karlsson

Insinöörityön tavoitteena on päivittää iOS-sovelluksen ohjelmointikieli ja rakenne, jotta se voitaisiin yhdistää toisen sovelluksen kanssa ja molempiin sovelluksiin voitaisiin kehittää ominaisuuksia tehokkaasti tulevaisuudessa. Hyödyntämällä uusia Xcode IDE:n ja Swift-ohjelmointikielen ominaisuuksia sovellus päivitetään tasolle, jossa sovellukseen tulevaisuuden muutokset ovat yksinkertaisempia implementoida.

Päivitettävässä sovelluksessa käytettiin pääasiassa Objective-C-kieltä eikä juurikaan Xcoden tarjoamia etuja. Toinen iOS-sovellus oli aikaisemmin päivitetty ja käytti Swift-kieltä ja oli kehitetty hyödyntämään Xcoden etuja. Sovellusten eroavaisuus aiheutti pakon uudelleen suunnitella ja luoda jokainen sovelluksien yhteinen ominaisuus kahdesti.

Projektissa käytettiin Xcode IDE:tä ja useita Applen tarjoamia ohjelmistokehyksiä uudelleen luomaan vanhat toteutukset. Versionhallinnassa käytettiin git-versionhallintaa ja Sourcetree-sovellusta.

Xcoden avulla sovellus päivitettiin hyödyntämään storyboard-tiedostoja näkymien luonnissa sekavan koodissa luodun näkymän sijasta. Ohjelmistokehysten ja Swift-koodin avulla yksinkertaistettiin useita ominaisuuksia, joiden vanhat ratkaisut olivat hyvin monimutkaisia ja muutosta vastustavia.

Tuloksena päivitettiin vanha Objective-C-kieltä käyttävä sovellus käyttämään Swift-kieltä ja iOS-kehityksessä hyödynnettäviä työkaluja. Tulevaisuudessa sovelluksen jatkokehittäminen ja uusien ominaisuuksien luonti on helpompaa.

Avainsanat	iOS, Swift, Objective-C

Author Title Number of Pages Date	Emil Itäjärvi Combination Project of iOS Applications 40 pages 26 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer Mia Karlsson, Instructor

The goal of the project was to update a mobile iOS app on the structural and code level utilizing the new features of Xcode IDE and the Swift programming language to update the application to a point where future changes would be simpler to implement.

The old app mainly used the Objective-C language and hardly any benefits offered by Xcode. Another iOS-app, on the other hand, had been updated earlier and now uses the Swift language and was updated to utilize the benefits offered by both the language and Xcode IDE. The differences between the two applications forced all the new designed features to be programed and implemented twice.

The project used Xcode IDE and several Apple frameworks to recreate the old features utilizing the new tools. Version control used git and Sourcetree.

With Xcode the app was updated to use storyboard files in creating and presenting views instead of the old way of programmatically crafting them. Frameworks and Swift code were used to simplify multiple features where the old methods were overly complex and resistant to change.

The result was an updated iOS-application that utilized the Swift language and multiple tools in its development. It is now easier and more convenient to update and develop new features for the application.

Keywords	iOS, Swift, Objective-C

Sisällys

Lyhenteet

IDE	Integrated Development Environment. Ohjelmistoympäristö. Ohjelma, jonka avulla ohjelmoija kehittää ohjelmia.
XML	Extensible Markup Language. Merkintäkieli, jonka avulla kuvataan usein rakennetta.
JSON	JavaScript Object Notation. Kevyt datan siirron formaatti, jonka avulla voidaan siirtää dataa ohjelmointikielestä riippumatta.
MVC	Model-View-Controller. Ohjelmistoarkkitehtuuri, joka erottaa käyttöliittymän sovelluksen tiedostoista.

1 Johdanto

Opinnäytetyöni aiheena on iOS-sovelluksen jatkokehitys ja sovelluksessa käytetyn Objective-C-ohjelmointikielen vaihto uudempaan Swift-ohjelmointikieleen. Lopputyö kehittää kyseisen projektin tasoon, joka helpottaa uusien ominaisuuksien kehittämistä. Muutoksien jälkeen projektin kehittäminen, korjaaminen ja muokkauksien suunnittelu kuluttavat vähemmän aikaa. Työn edetessä tutustutaan iOS-kehityksessä käytettäviin ohjelmointikieliin, sovelluksiin ja kehitysmahdollisuuksiin.

Opinnäytetyön toimeksiantajana toimii yritys, joka heidän toiveidensa mukaan pysyy tuntemattomana. Yrityksellä on kaksi sovellusta, joilla on osittain samanlaiset ominaisuudet. Aikaisempien sovelluskehityspäätösten toimesta sovelluksien jatkokehitys on hidastunut ja uusien ominaisuuksien luonti hankaloitunut. Yksi sovellus on päivitetty käyttämään Swift-ohjelmointikieltä ja uudempia kehitysmenetelmiä. Kyseiseen sovellukseen viitataan lopputekstin ajan nimellä projekti A. Projekti A on muokattu versio yrityksen toisesta sovelluksesta. Tämä toinen sovellus oli alkuperäinen versio ja tähän sovellukseen viitataan opinnäytetyön aikana nimellä projekti B. Projekti B on vanha sovellus, eikä se ole saanut päivitystä Objective-C-kielestä Swiftille, mikä hyödyntää vielä vanhoja kehitysmenetelmiä.

Opinnäytetyön alkuna oli yrityksen ongelmallinen tilanne. Ongelmat ovat ajallisesti ja työn määrän kannalta haastavat. Uusia ominaisuuksia ei voitu kehittää molemmille sovelluksille samaan aikaan, koska ohjelmointikielet ja projektien rakenteet olivat liian erilaiset. Kehittäessä uusia ominaisuuksia sovelluksille kehittäjien piti osata kahta eri kieltä. Vaikka uusi ominaisuus luotaisiin, suora yhdistäminen olisi mahdotonta ilman kielen kääntämistä tai Objective-C/Swift-kielten yhdistämisoperaatiota. Opinnäytetyön tarkoituksena on korjata nämä ongelmat suorittamalla useita päivityksiä projekti B:hen. Projektin vanha Objective-C-koodi päivitetään käyttämään projekti A:n Swift-kieltä ja hyödyntämään molemmissa projekteissa käytettävän Xcoden kehitysominaisuuksia. Ohjelmistokielen kääntämisen jälkeen projektit A ja B voidaan yhdistää toisiinsa projektien yhteisten osuuksien osalta. Sovelluksista ei tule yksi sovellus vaan molempien erot säilyvät Xcoden ominaisuuksien ansiosta.

2 Ohjelmointikielet

Projekteissa käytettävät ohjelmointikielet ovat Swift ja Objective-C. Swift on ohjelmointikielenä uusi, projekti B käännetään kyseiselle kielelle. Objective-C on vanhempi ohjelmointikieli, josta johdannon mukaan on tarkoitus luopua ja yhdistää sovelluksien toimivuudet.

2.1 Objective-C

Objective-C on yleiskäyttöinen olio-ohjelmointikieli, joka toimi Applen pääasiallisena ohjelmointikielenä heidän käyttöjärjestelmiensä kehityksessä. Oleellisin osa iOS-käyttöjärjestelmään ja sen eri ohjelmakehyksiin liittyen on, että kaikki käyttävät Objective-C-elinkaarta. Tämä mahdollistaa kielituen C-, C++- ja Objective-C-ohjelmointikielille. Uusimpana ohjelmointikielilisänä on Swift.

2.2 Swift

Swift on hyvin nuori ohjelmointikieli, jonka kehitti Apple ja se julkaistiin 2014 Applen Xcode 6 -version mukana. Se on hyvin yleiskäytännöllinen ohjelmointikieli, sillä sen ohjelmointiparadigma on monimuotoinen tukien funktionaalista, olio-, protokolla- ja imperatiivista ohjelmointia. Swiftin tarkoituksena on tukea monia Objective-C-kielestä löytyviä pääominaisuuksia, mutta paljon turvallisemmin ja nopeammin toteutuksessaan. Tuloksena on ohjelmointikieli, joka hyödyntää moderneja ratkaisuja ja ominaisuuksia avustaakseen kehittäjien työtä. [1.]

Objective-C-kieleen verrattuna Swift on paljon kehittäjäystävällisempi. Swift tukee päätettyjä tyyppejä tehdäkseen koodista puhtaampaa ja vähentää todennäköisyyttä virheille. Samalla se käsittelee automattisesti muistin välttäen C-kielelle tunnetut muistin hallinnointiongelmat. Swiftillä on myös paljon helpompi käsitellä arvoja, koska sen moduulit tarjoavat nimiavaruuden ja niille voidaan antaa ”var”, muokattava muuttuja tai ”let”-vakiomuuttuja. Swift myös yrittää välttää yleistä tyhjää nil-arvoa, mikä aiheuttaa usein ohjelmissa kaatumisia vaatimalla arvon muuttujille. Tietyissä tapauksissa tyhjä arvo voi olla

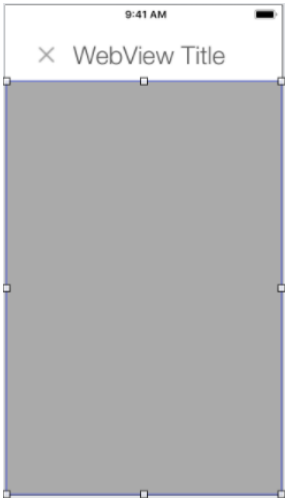
tarpeellinen, joten Swift tarjoaa mahdollisuuden tehdä objektista tyhjän ”optionals”-ominaisuuden avulla. Optional-arvot voivat sisältää nil-arvon ja Swift-syntaksi pakottaa käyttämään ?-merkkiä, joka ilmoittaa kehittäjälle ja ohjelmalle, että arvo voi olla tyhjä. Swiftistä on yritetty tehdä yleisesti helppokäyttöiseksi ja sopivaksi monipuolisiin tilanteisiin. [2.]

3 Xcode-kehitysympäristö

Xcode on macOS:lle kehitetty ohjelmointiympäristö, joka on tarkoitettu Applen käyttöjärjestelmillä toimivien tuotteiden ohjelmointikehitykseen. Ensimmäinen versio Xcode 1.0 julkaistiin vuonna 2003 ja perustuu Project Builder IDE:hen [3]. Uusin versio tällä hetkellä on 11, joka julkaistiin vuoden 2019 puolessa välissä. Xcode sisältää oleellisia ja käteviä sovelluskehitystyökaluja, joita käytetään projektin päivityksessä. [4.]

3.1 Storyboard-ominaisuus

Säilyttääkseen näkymiä hyvässä järjestyksessä Xcode tarjoaa visuaalisen esityksen sovelluksen näkymistä. Vaikka sovellus ei suoranaisesti vaadi storyboardin olemassaoloa, se on todella hyödyllinen työväline kehittäjille havainnollistamaan näkymien suhteita toisiinsa ja keräämään valitut näkymät samaan storyboardiin. Storyboard-tiedosto voidaan muokata helposti Interface Builderin kanssa, josta kerrotaan myöhemmässä luvussa. Kuvassa 1 esitetään yksi storyboard-tiedoston näkymistä eli sceneistä, jotka muodostavat pääosin sovelluksen visuaaliset elementit. Näkymässä (kuva 1) esiintyy eri elementtejä, kuten X-merkki on UIButton ja WebView Title on UILabel, jonka tekstiksi on määritelty WebView Title.



Kuva 1. Storyboard Scene visuaalisessa muodossa.

Storyboard sisältää scenejä eli näkymiä, jotka ovat liitoksissa näkymien hallitsijoihin tai niihin liitoksissa oleviin näkymiin. Storyboard muodostuu yhdestä tiedostosta, joka on kirjoitettu XML-ohjelmointikielellä ja esitetään visuaalisesti kehittäjälle. Halutessa voidaan katsoa tiedostoa XML-koodina visuaalisen esityksen sijasta. [5.]

```
<!--Web Menu View Controller-→
     <scene sceneID="g9V-Da-obL">
               <objects>
                    <viewController storyboardIdentifier="WebMenu" id="yFy-
                    wK-rGo" customClass="WebMenuViewController" customModu-
                    leProvider="target" sceneMemberID="viewController">
```

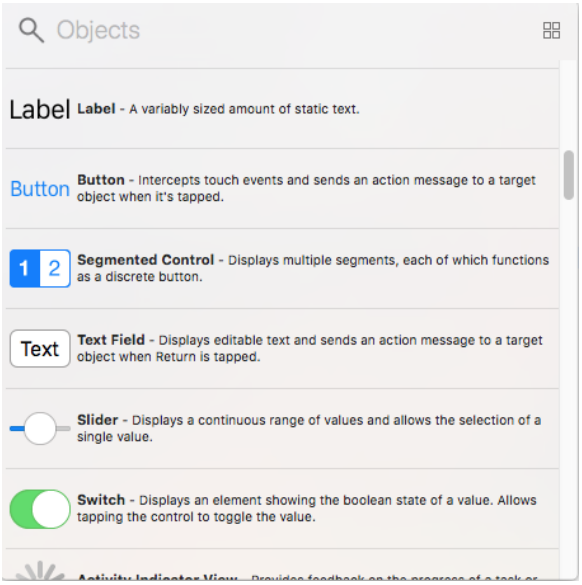
Esimerkkikoodi 1. XML-koodi jakaa jokaisen scenen omaan osioon

Esimerkkikoodissa 1 on pieni osa XML-koodia, joka luo osan kuvan 1 näkymästä. Koodissa määritellään sceneID, jonka avulla näkymät erotetaan toisistaan. Objects-osio sisältää kaikki elementit, joita näkymästä löytyy. Storyboard identifier määrittelee, millä nimellä näkymä löytyy storyboardista ja millä nimellä sitä koodin puolella kutsutaan. Näkymää hallinnoi mukautettu luokka nimeltään WebMenuViewController. Jokaiselle näkymälle voidaan antaa oma mukautettu luokka, joka on vastuussa näkymän esittämisestä, muokkauksesta ja käyttäjän syötteiden prosessoinnista. Viewcontrollereissa suoritetaan suurin osa näkymien toiminnasta, mutta storyboardissa näkymien suhteisiin voidaan viitata Seguen avulla. Seque linkittää kaksi näkymää toisiinsa ja koodin puolella suorittaa

yksinkertaisen näkymän vaihdon käskystä. Seque ei ole pakollinen osuus näkymiä tai sovellusta, mutta hyödyllinen näkymien organisoinnissa. [6.]

3.2 Interface Builder -ominaisuus

Storyboardissa olevien näkymien visuaaliseen muokkaukseen käytetään Xcodesta löytyvää Interface Builder -editoria. Kuten nimi antaa ymmärtää Interface Builderia käytetään rakentamaan ja muokkaamaan sovelluksien näkymiä. Kuvassa 2 esitetään, miten uusien näkymien luonti ja muokkaus on yksinkertaista tällä editorilla, koska voit valita erilaisia valmiiksi rakennettuja ikkunoita, nappeja, tekstikenttiä tai yleisesti määriteltyjä elementtejä ja lisätä ne haluttuun näkymään. [7.]



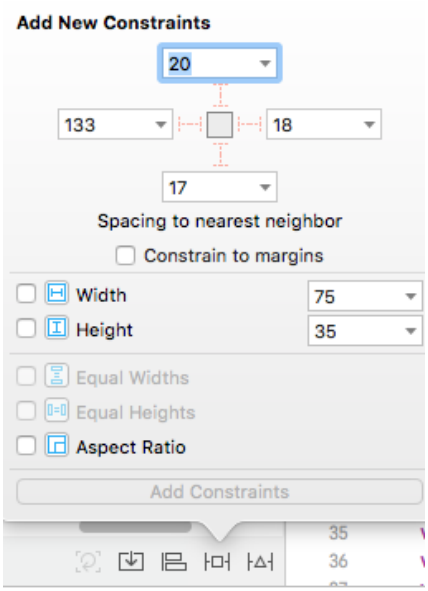
Kuva 2. Yleisiä elementtejä on luotu valmiiksi käyttöä varten.

Jokaisella elementillä on omia valmiiksi määriteltyjä muuttujia, joita voidaan muokata Interface Builderin näkymässä. Esimerkiksi UIButton-elementillä on oma teksti, kuva, tausta ja muita vastaavan tason muuttujia, joiden avulla nappia voidaan muokata halutun muotoiseksi. Jokaiseen elementtiin kuuluu myös XY-koordinaatiston sijainti-, pituus- ja leveysmitat. Näkymään voidaan kehittää myös dynaamisuutta lisäämällä rajoitteita, jotka

määrittelevät yhden elementin suhteen muihin elementteihin. Rajoite on esimerkiksi se, kuinka kaukana elementti on sen yläpuolella olevasta elementistä.

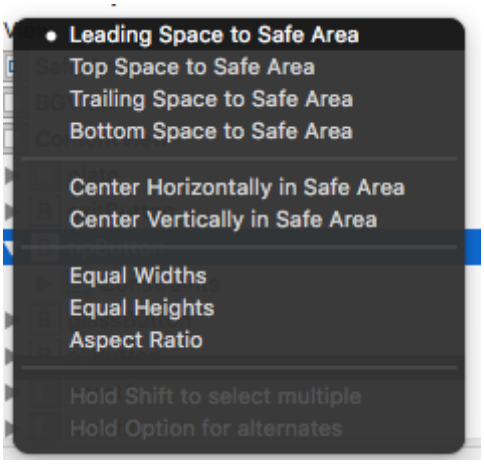
3.3 Auto Layout -ominaisuus

Auto Layout on työkalu näkymässä esiintyvien kappaleiden sijaintien määrittelyyn. Näkymän kappaleisiin voidaan asettaa Auto Layoutin avulla eri rajoitteita, jotka dynaamisesti laskevat kappaleen koon ja sijainnin näkymässä. Kuvan 3 tapaan voidaan määritellä kappaleen olevan 133 pistettä näkymän vasemmasta kulmasta ja 18 pistettä oikeasta. Samalla kertoen, että kappale tulee olemaan aina 20 pistettä korkea.



Kuva 3. Rajoitteita voidaan asettaa manuaalisesti.

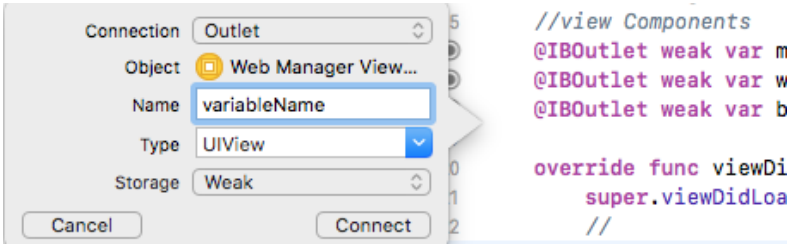
Kuvassa 3 esitettyyn rajoitteisiin perustuva ratkaisu kehitykseen mahdollistaa dynaamisen muutoksen perustuen sisäisiin ja ulkoisiin elementteihin. Rajoitteet voidaan tehdä myös näkymässä hyödyntäen hiirtä kiskoen elementistä sinertävän viivan toisen elementin päälle Ctrl + hiiren vasemman napin avulla, mikä avaa käyttäjälle kuvassa 4 näkyvän ikkunan.



Kuva 4. Rajoitteita voidaan asettaa myös näkymässä.

Kuvan 4 Safe Area on alue, jonka sisällä on turvallista esittää sovelluksen sisältöä ilman, että sovellus ilmestyisi esimerkiksi kännykän kellon päälle. Tämän takia Safe Area tarjotaan jokaisessa näkymässä, jos Safe Area -ominaisuus on käytössä. Rajoitteet muokkaantuvat ulkoisten ja sisäisten muutoksien vaikutuksesta. Ulkoisia muutoksia ovat esimerkiksi ikkunan koon muuttaminen Macillä tai kännykän kääntäminen vaakasuuntaisesta asennosta pystysuuntaiseen. Sisäiset muutokset ovat pääasiassa näkymään kohdistuvia muutoksia, jotka tapahtuvat yhdessä koodin kanssa: esimerkiksi lokalisaatiot tai näkymän sisällön muuttuminen käyttäjän syötteiden toimesta. [8.]

Kaikki Auto Layoutilla luodut rajoitteet voidaan myös liittää koodiin ja niiden arvoja voidaan muuttaa koodin puolella eri tavoin. Jokainen näkymän elementti ja niiden rajoitteet voidaan yhdistää koodiin muuttujiksi helposti, kuten kuvassa 5 näkyy.



Kuva 5. WebManager näkymän UIView-elementti yhdistetään näkymänhallitsija tiedostoon IBOutlet-muuttujaksi, jota sitä voidaan muokata koodin puolella.

Kuvassa 5 WebManager näkymän UIView-elementti yhdistetään sen UIViewControlleriin nimeltään WebManagerViewController ja UIView-elementistä tehdään IBOutlet-muuttuja. Tämän ansiosta UIView-elementtiin voidaan tehdä muokkauksia koodin puolella. Suurin osa näkymään vaikuttavista muutoksista voidaan tehdä suoraan Interface Builderin avulla, koska Apple kannustaa kehittäjiä hyödyntämään MVC-arkkitehtuuria sovelluskehityksessä. Näkymät voidaan rakentaa kokonaan irrallisina koodista, mutta kehityksessä voi olla tilanteita, joissa koodin puolella halutaan muokata näkymää. Koodissa muokkaus voi tulla oleelliseksi, jos halutaan muokata tiettyä näkymää riippuen sen sisällöstä tai kännykkämallista. Jos käsitellään kuvia ja niiden kokoa ei ole tarkalleen määritelty, voidaan tarvita dynaamisia muutoksia, jotta jokainen kuva pysyy omassa mittasuhteessaan eikä vie tarpeetonta tilaa näkymästä. Toinen käyttötarkoitus muokkaukseen voi olla lokalisaatiot, jotka saattavat tarvita muutoksia näkymään esittääkseen saman materiaalin eri kielillä.

Riippumatta miten paljon Auto Layoutin ominaisuuksia halutaan hyödyntää tai muokata, se takaa paljon helpomman ja visuaalisesti selvän menetelmän rakentaa uusia näkymiä ja määritellä niiden elementtien sijaintia näkymässään. [9.]

Kuva 6. 2. nappi (tumma) ja sen oikein puolinen nappi antaa kehittäjän vaihtaa yhden näkymän ja kahden näkymän välillä. Kehittäjä voi siis tarkistella Interface Builderia ja koodia samaan aikaan.

Yhdistämällä halutun näkymän ja sen näkymänhallitsija kuvan 5 sivulla 7 tapaan ja avaamalla molemmat osat kuvan 6 esittämällä tavalla vierekkäin kehittäjä näkee sovelluksen kokonaisuudessaan. Koko kuvan havainnollistaminen avustaa sovelluksen suunnittelussa ja rakentamisessa.

3.4 Ohjelmistokehykset

Sovelluksen kehitys vaatii useita eri elementtejä ja kokonaisuuksia, minkä takia Apple tarjoaa apua erilaisilla valmiilla kokonaisuuksilla. Frameworkit eli ohjelmistokehykset

ovat oleellinen osa jokaista vähääkään monimutkaista sovellusta. Apple tarjoaa monia ohjelmistokehyksiä, joiden tarkoituksena on avustaa sovelluksen kehittäjää eri alueissa. Xcode on tehnyt näiden kätevien työkalujen lisäämisestä hyvin suoraviivaista, mutta on olemassa hyödyllinen riippuvaisuusmanageri nimeltään Cocoapods, joka on otettu käyttöön tässä projektissa. Cocoapods luo pod-tiedoston, joka määrittelee, mitä ohjelmistokehyksiä ja muita kirjastoja hyödynnetään projektissa ja mikä jokaisen versionumero on. [10.]

3.4.1 Foundation-ohjelmistokehys

Foundation on yksi tärkeimmistä ohjelmistokehyksistä, koska se tarjoaa kriittisiä ominaisuuksia ja toiminnallisuuksia sovelluksille ja muille ohjelmistokehyksille. Näihin ominaisuuksiin kuuluvat tiedon tallennus ja säilyvyys, tekstin prosessointi, päivämäärä ja aikalaskennat. Foundation tarjoaa myös useita luokkia, protokollia ja datatyyppejä, joita voidaan käyttää macOS-, iOS-, watchOS- ja tvOS-käyttöjärjestelmissä. Jokainen luokka, joka haluaa työskennellä edes yksinkertaisten ominaisuuksien parissa, pitää tuoda Foundation-ohjelmistokehys mukanaan. Ilman Foundationia sovelluksella ei olisi pääsyä käsiksi olennaisiin ominaisuuksiin kuten:

- numeroarvo (int, decimal ja double)
- string ja attributedstring
- kokoelmat (array, dictionary ja set)
- aika (date, timeInterval, calendar).

Lisäksi Foundation tarjoaa monia muita ominaisuuksia, jotka ovat todella tärkeitä sovelluskehityksen kannalta. Ohjelmistokehyksen avulla pääset seuraaviin ominaisuuksiin käsiksi:

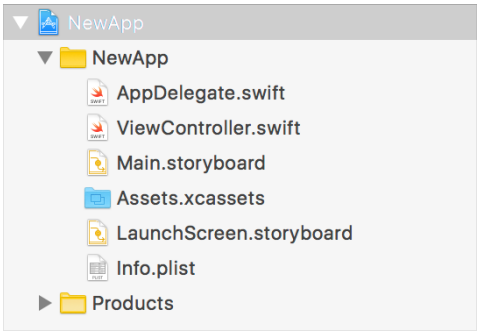
- tiedostojärjestelmä

- preferenssit
- iCloud
- URL-lataussysteemi
- ilmoitukset
- virheet ja poikkeukset
- resurssien hallintatemplate.

Nämä ovat tärkeitä ominaisuuksia sovelluksen kehityksessä ylipäätään, tässä opinnäytetyössä näitä ominaisuuksia hyödynnetään paljon. [11.]

3.4.2 UIKit-ohjelmistokehys

Näkymät ovat pakollinen osa sovelluksia ja Apple on kehittänyt ohjelmistokehyksen avustaakseen näkymien kanssa. UIKit on oleellinen osa käyttöliittymän luontiprosessia. Foundation ja UIKit molemmat tuovat oleellisia kokonaisuuksia kehittäjälle. Näiden kokonaisuuksien avulla voidaan esittää sovellus käyttäjälle, mahdollistaa sovelluksen käyttö ja hallinnoida käyttäjän syöttämää tietoa. [12.]



Kuva 7. Xcode luo valmiita tiedostoja Single View app -mallissa [12. Kuva 1.].

Kuvassa 7 sisällä voidaan nähdä jo muutamia UIKit-ohjelmistokehyksen tuomia kokonaisuuksia. LaunchScreen.storyboard pitää hallussaan käyttäjän ensimmäisen näkymän. Tämä esitetään käyttäjälle joka kerta, kun sovellus avataan ja kertoo käyttäjälle, että sovellus on avautumassa. Assets.xcassets on vastuussa esimerkiksi kuvien tallentamisesta. Näihin kuviin kuuluu myös sovelluksen ikoni App Storessa ja käyttäjän puhelimessa. Suurin osa näkymien käsittelyyn liittyvästä työstä tapahtuu ViewController.swift- ja Main.storyboard-tiedostoissa. ViewController.swift perii näkymän hallitsijan ja sen avulla pystyy käsittelemään ViewController.swiftille määriteltyä näkymää. Main.storyboard on pääasiallinen näkymien varasto, josta mainitaan enemmän Storyboard-luvussa.

UIKitin monipuolisin osuus on näkymien hallitsijat, joiden vastuulla on näkymien muokkaus ja käyttäjän toimintojen vastaanotto. Näkymän hallitsijoita on monia ja jokainen suorittaa eri tehtäviä, ne jaetaan kahteen eri ryhmään. Näkymän sisällön hallinnoijat ovat

- UIViewController
- UITableViewController
- UICollectionViewController.

Näiden vastuulla on käsitellä näkymien sisällä olevia elementtejä ja niihin liittyviä interaktioita. Yleisin näistä hallitsijoista on UIViewController, joka suorittaa yleisesti näkymän käsittelyn. UITableViewController on erikoistunut taulukkonäkymien hallinnointiin, UICollectionViewController on erikoistunut hallinnoimaan kokoelmia. Toinen ryhmä on vastuussa enemmän näkymien vaihtamisesta ja näihin näkymien hallitsijoihin kuuluvat:

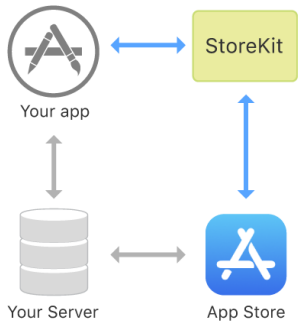
- UINavigationController
- UITabBarController
- UIPageViewController

- UISplitViewController.

UINavigationController ja UITabBarController toimivat samalla tavalla, eli esittävät ja käsittelevät navigointiin tarkoitettua palkkia. UINavigationController on yleensä näkymän yläosassa ja voi sisältää eri toimintoja suorittavia nappeja. UITabBarController toisaalta on näkymä, joka muodostuu radionapeista, jotka vaihtavat näkymiä ja sijoittuvat yleensä ruudun alaosaan. UIPageViewController ja UISplitViewController ovat samanlaisia toiminnoiltaan. UIPageViewController esittää useita näkymiä, joita sille on annettu. Pyyhkäisemällä haluttuun suuntaan UIPageViewController käsittelee näkymän vaihdon. UISplitViewController toisaalta esittää nimensä mukaisesti näkymiä vierekkäin ja sitä voidaan hyödyntää esimerkiksi tutkiessa listan objekteja, joilla on oma näkymä ja listan objektin näkymää halutaan esittää samanaikaisesti. Molemmat ryhmät ovat tärkeitä sovelluksen kehitykseen liittyviä työkaluja ja tässä opinnäytetyössä hyödynnetään kyseisiä näkymän hallitsijoita luotaessa monia sovelluksen eri ominaisuuksia. [13.]

3.4.3 StoreKit-ohjelmistokehys

Sovelluksessa tulee olemaan sovelluksen sisäisiä tuotteita, joita myydään asiakkaille. Tuotteiden myyntiin tarvitaan ostotapahtuman käsittely ja tuotteen yhdistäminen käyttäjän Apple-tiliin. StoreKit on Applen ohjelmistokehys, joka mahdollistaa yhteyden App Storen ja sovelluksen välillä. Sovelluksen monetisaatio-malli perustuu tähän kyseiseen ohjelmistokehykseen ja sen avulla käsitellään sovelluksen sisäiset ostot ja kuittien tarkistukset. StoreKit-ohjelmistokehys toimii välikätenä sovelluksen ja App Storen kanssa, kuten kuvassa 8 näkyy ja samalla mahdollistetaan turvallinen ostotapahtuma. [14.]



Kuva 8. StoreKit toimii välikätenä sovelluksen ja App storen kanssa [15. Kuva 1].

StoreKitin avulla Apple suorittaa kaikki sovelluksen sisällä ostettujen tuotteiden maksuprosessin turvallisesti ja palauttaa sovellukselle kuitin todistuksena ostosta. Sovelluksella voi olla useita eri tuotteita tarjolla, mutta kaikki jakautuvat neljään tuotetyyppiin.

- kuluvat tuotteet
- ei kuluvat tuotteet
- automaattisesti uusiutuvat tuotteet
- ei automaattisesti uusiutuvat tuotteet.

Ei-kuluvat tuotteet ja automaattisesti uusiutuvat tuotteet on suunniteltu pysymään käyttäjällä määrätyn ajan riippumatta, millä laitteella sovellus on. StoreKit pystyy kommunikoimaan App Storen kanssa, jotta kuittien avulla voidaan palauttaa sovelluksen käyttäjälle heille kuuluvat tuotteet. Kommunikointiin tarvitaan StoreObserver ja SkPaymentQueue. StoreObserver on luokka, joka perii SKPaymentTransactionObserverin ja on oleellinen osa seuraamaan App Storen ja StoreKitin keskustelua sovelluksen kanssa. SKPaymentQueue käsittelee ostosten käsittelyn ja tarvitsee StoreObserverin toimiakseen (esimerkkikoodi 2). [15.]

```
let iapObserver = StoreObserver()
SKPaymentQueue.default().add(iapObserver)
```

Esimerkkikoodi 2. StoreObserver annetaan SKPaymentQueuelle, jotta maksutapahtumat voidaan suorittaa.

Ostotilanteessa SKPaymentQueuelle annetaan tuotteen ID samalla add-funktiolla, ostotilanne astuu seuraavaan vaiheeseen. Tuotteen oston yhteydessä aikaisemmin lisätty StoreObserver saa tiedotteen App Storesta tuotteen ostosta ja kuitin todisteena. App delegaten didFinishLaunchingWithOptions function käynnistyessän on tärkeää antaa SKPaymentQueuelle StoreObserver, jotta sovellus voi kommunikoida App Storen kanssa (esimerkkikoodi 3).

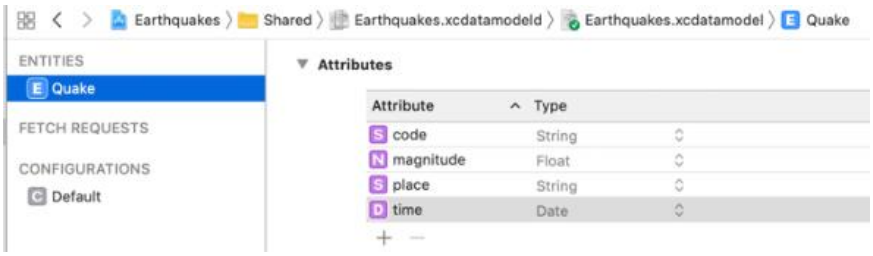
```
class AppDelegate: UIResponder, UIApplicationDelegate {
      func application(_ application: UIApplication, didFinishLaunchingWithOp-
      tions launchOptions:      [UIApplicationLaunchOptionsKey: Any]?) -> Bool
      {
            SKPaymentQueue.default().add(iapObserver)
            return true
      }
      func applicationWillTerminate(_ application: UIApplication){
            SKPaymentQueue.default().remove(iapObserver)
      }
```

Esimerkkikoodi 3. Applen dokumentaatiossa esitetään, miten AppDelegatessa suoritetaan aikaisempi toimenpide (esimerkkikoodi 2) sovelluksen käynnistyessä.

AppDelegate on tärkeä osa yleistä sovelluksen rakennetta ja on vastuussa monista sovelluksen käynnistykseen ja sulkeutumiseen liittyvistä osista. Applen vaatimuksien mukaan StoreObserver tulee myös poistaa sovelluksen sulkeutuessa applicationWillTerminate-funktiossa, kuten esimerkkikoodissa 3 näkyy. [16.]

3.4.4 Core Data -ohjelmistokehys

Tiedon tallentaminen sovelluksen sisällä on tärkeää, jonka takia Core Data on yksi oleellisimmista ohjelmistokehyksistä. Tämä ohjelmistokehys mahdollistaa sovellukselle pysyvän tiedon säilytyksen ja välimuistin käytön. Core Datan tiedonhallintaeditorin avulla voidaan määritellä eri tietotyyppejä, suhteita ja luoda luokkamääritelmiä. Oleellisesti Core Data ei ole riippuvainen Objective-C- tai Swift-kielestä ja pystyy tallentamaan ja lukemaan tietoja molemmille ohjelmointikielille. Datamalli sisältää kokonaisuuksia, jotka muodostuvat eri attribuuteista (kuva 9).



Kuva 9. Apple-dokumentaatiossa esiintyvä esimerkki datamallista [16. Kuva 1].

Datamallit voivat säilyttää monipuolista informaatiota sovelluksen kappaleista. Tämä xcdatamodel-tiedontallennusmenetelmä on verrattavissa SQL- ja noSQL-tietokantoihin.

Lopputuloksena on joustava tietokanta, joka toimii riippumatta ohjelmointikielestä ja jota pystytään muokkaamaan tarpeiden mukaan. [17.]

3.4.5 Core Graphics -ohjelmistokehys

Core Graphics hyödyntää Quartzin teknologiaa luodakseen kevyitä 2D-renderöintejä. Tämän ohjelmistokehyksen avulla voimme luoda polkuihin perustuvia piirustuksia, hallinnoida värejä, käsitellä kuvadataa tai luoda PDF-dokumentteja. CG tarjoaa monia geometrisiä datatyyppejä kuten:

- CGFloat
- CGPoint
- CGSize
- CGRect.

Nämä ovat oleellisia osia näkymien muokkauksessa, sillä monet UIKitin avulla luodut näkymät voivat ottaa vastaa näihin datatyyppeihin perustuvaa dataa. CGRect esimerkiksi sisältää lähtöpisteen X/Y-koordinaatistossa ja kappaleen koon width- ja height-arvoina. Näkymien muokkauksen lisäksi CG tarjoaa 2D-manipulaatiolle tärkeitä luokkia kuten:

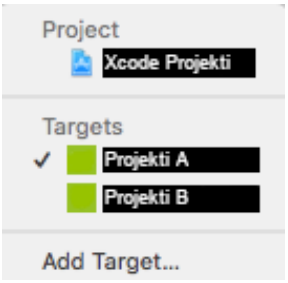
- CGContext
- CGMutablePath
- CGLayer.

CGContext on monipuolisin, sillä sen avulla koodin puolella voidaan piirtää näkymään eri kuvioita. CGContextille voidaan määritellä, mitä piirretään: esimerkiksi linja, suorakulmio tai ympyrä. CGContextille voidaan ilmoittaa myös halutusta väristä, viivan

paksuudesta tai mahdollisesta varjosta tarjoten monipuolisia vaihtoehtoja koodin puolella suoritettavissa piirroksissa. CGMutablePath ja CGLayer ovat tärkeämpiä maskien luonnissa, mikä mahdollistaa esimerkiksi isojen kuvien laskelmoidun leikkauksen. [18.]

3.5 Sovellusten erottelu

Opinnäytetyön aikana projekti A ja projekti B tullaan yhdistämään yhteisten osien avulla. Sovellukset halutaan kuitenkin pitää erillisinä. Sovellusten erottelussa käytetään build targetia, joka määrittelee rakennettavan ohjelman ja siihen liittyvät tiedot. Jokainen sovellus määritellään kahdella tasolla, ensin projektitasolla, jolla määritellään yleisemmin koko sovellus. Projektitason määrittely ei kerro yksittäiselle sovellukselle tärkeitä tietoja, jonka takia sovellus määritellään myös build target -tasolla. Jokainen target perii suurimman osan arvoistaan projektilta, jos targetille ei erikseen toisin määritellä. Targeteilla on kuitenkin täysin omat muuttujat, joiden avulla käyttöjärjestelmä erottaa sovelluksen toisesta sovelluksesta. Projektitiedosto tai Target-tiedostot voidaan helposti valita niille luodusta listasta (kuva 10).



Kuva 10. Target-listassa voidaan nähdä, mitä build targeteja on projektissa. Haluttu target voidaan myös valita editoitavaksi.

Target-tiedostoilla on useita paneeleita, joissa määritellään eri tietoja Targetin toiminnasta. Pääasiassa keskitytään General-paneeliin, jossa kerrotaan sovellukselle tärkeimmät tiedot tässä projektissa. Riippuen käyttöjärjestelmästä, johon sovellus on suunniteltu kehitettäväksi, muutamat kohdat tällä paneelilla muuttuvat. Koska projekti on iOS-sovellus ja aiheena on siihen liittyvät osuudet, keskitytään vain siihen liittyvien tietojen selitykseen. General-paneeliin kuuluvat aina seuraavat arvot: bundle-identiteetti, versionumero, rakennusnumero, Apple Developer Program -kehittäjätiimi, kehitysversionkohde,

lista laitteista, joilla sovellus toimii, aloitusnäkymä ja sovelluksen näkymän orientaatio. Projektille merkittävin tieto on bundle-identiteetti, koska sen avulla käyttöjärjestelmä ja App Store tunnistavat sovelluksen. [19.]

Xcode-projektissa targetien välinen erottelu tapahtuu tiedostojen ja koodin tasolla. Tämä luo organisointia sovellukselle ja on oleellinen ominaisuus hallinnoimaan sovelluksen rakennetta. Erottelu on tärkeää varsinkin tilanteissa, joissa sovelluksessa on tiedostoja, joiden erottelu koodin puolella tulee mahdottomaksi. Näihin kuuluvat esimerkiksi lokalisaatiotiedostot. Tämä erottelu on mahdollista tiedostojen Target Membership -muuttujan avulla (kuva 11).

Kuva 11. Jokaisella tiedostolla on Target Membership -muuttuja, jossa merkitään, mihin targetiin kyseinen tiedosto kuuluu.

Kuvassa 11 valittu tiedosto on merkitty olevan vain tietyn targetin käytössä, jolloin toinen target ei ole mitenkään tietoinen valitun tiedoston olemassa olosta. Target Membership voidaan määrittää usealle tiedostolle ja tiedosto voidaan jakaa molemmille targeteille. On tärkeää huomioida, miten tiedoston olemassaolo vaikuttaa sovelluksen toimintaan. Esimerkkinä molemmat targetit hyödyntävät yhteistä tiedostoa, mutta kyseiselle tiedostolle tehdään koodin puolella muutoksia riippuen siitä, kumpi target on kyseessä (esimerkkikoodi 4). [20.]

```
#if CUSTOMFLAGFORTARGETB
     print(”hello world”)
#else
     print(”world hello”)
#endif
```

Esimerkkikoodi 4. Koodissa käytetään build targetille annettua custom flag-arvoa: jos arvo löytyy, if-lauseen sisällä oleva koodi toteutetaan.

Esimerkkikoodissa 4 yhdessä tiedostossa suoritetaan kaksi eri koodia riippuen siitä, mistä targetista on kyse. Jos targetia ei oteta huomioon ja koodissa yritetään kutsua

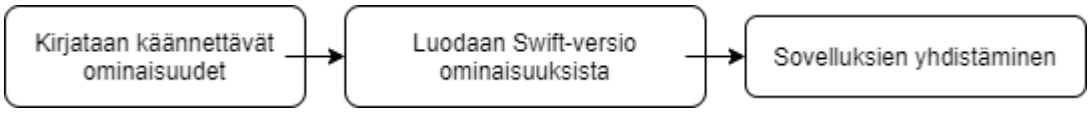
toista tiedostoa, jota target ei tunne, aiheuttaa tämä sovelluksen kaatumisen. Joissain tapauksissa tiedostojen jakaminen on turhaa, mutta tiedostojen kuten kuvien ja lokalisaatioiden kanssa tämä voi olla pakollista. Kuvassa 12 lokalisaatiotiedosto on jaettu targetien mukaan.

Kuva 12. Lokalisaatiotiedostot jaetaan kahden sovelluksen kesken.

Lokalisaatiotiedostojen jako targetien mukaan opinnäytetyön projektissa on tarpeellista, koska projekteilla on eri lokalisaatioita. Kuvassa 12 näkyy, miten Swift-kielinen projekti A on käännetty usealle kielelle ja tämän takia kaikki 5 ensimmäistä lokalisaatiotiedostoa on merkitty Target Membership -arvoksi projekti A. Projekti B:stä puuttuu osittain ruotsin-, saksan- ja espanjankieliset lokalisaatiot, jonka takia projektin targetille esitetään vain kaksi lokalisaatiotiedostoa.

3.6 Projektin vaiheet

Opinnäytetyön kääntämisprojekti voidaan jakaa kolmeen osaan, jotka ovat määrittely, kääntäminen ja yhdistäminen. Ensimmäisessä osassa määritellään, mikä on projektin tavoite ja mitkä osat vanhasta Objective-C-projektista tulee kääntää Swiftille. Toisessa vaiheessa käännetään ensimmäisessä vaiheessa valitut osat Swiftille. Viimeisessä vaiheessa sovellukset yhdistetään toisiinsa, niiden yhteisten ominaisuuksien pohjalta. Kuva 13 esittää jokaisen vaiheen järjestyksessä, se on hyvin oleellista, ettei mitään näistä vaiheista jätetä väliin.



Kuva 13. Projekti voidaan jakaa kolmeen vaiheeseen.

Työn tehostamiseksi on myös tärkeää, että käydään läpi, mitä osuuksia ei ole tarpeellista kääntää, koska ne on jo aikaisemmin käännetty Objective-Cstä Swiftille. On myös huomioitava, ettei Swift-projektissa kaikkia osuuksia ole käännetty Objective-C-kielestä Swiftille, koska näiden osuuksien uudelleen työstäminen olisi liian suuri työtehtävä kääntämisestä saataviin hyötyihin nähden ja Objective-C-versiot toimivat ongelmitta nykyisen Swift-toteutuksen kanssa. Lisäksi on tärkeää ymmärtää, että Swift ja Objective-C voivat toimia yhdessä ja Swift on kehitetty mahdollisimman joustavaksi tässä aspektissa. Ennen kääntämisvaiheen aloittamista kaikki ominaisuudet listataan ja niistä kerätään oleelliset tiedot. Nämä tiedot voidaan jakaa neljään ryhmään (kuva 14).

Kuva 14. Kuvassa on visuaalinen esitys ominaisuudesta kerätystä tiedosta. Tämä määrittelee ominaisuuden ja sen toiminnallisuudelle tärkeät osuudet.

Kuvassa 14 jokaiselle ominaisuudelle tärkeät osat jaetaan neljään eri ryhmään: tarvitut alustavat tiedot, näkymä, näkymänhallitsija ja muut luokat. Alustaviin tietoihin kuuluu kaikki tiedot, joita ominaisuus tarvitsee näkymän ja näkymänhallitsijan alustamisessa. Näkymään kuuluu kaikki näkymän elementit, jotka esiintyvät ominaisuuden näkymässä. Näkymänhallitsija käsittelee, miten näkymä toimii ominaisuuden käytön aikana. Muut luokat-osioon kuuluvat sovelluksen osat, jotka eivät ole näkymänhallitsijassa, mutta ovat siinä käytössä. Näitä on esimerkiksi sovelluksessa käytettävä LoadingView-luokka, joka on vastuussa sovelluksessa käytettävästä latausruudusta. Kun kaikki halutut sovelluksen ominaisuudet on luokiteltu ja niiden riippuvuudet on listattu, voidaan katsoa isompaa

kuvaa ja harkita, miten lähdetään eri osia kääntämään. Tähän päätöstyöhön vaikuttaa suuresti ominaisuuden tarkoitus ja siihen liittyvät osat. Esimerkiksi, jos ominaisuus vaatii näkymän, voidaan hyödyntää Xcoden Interface Builderia ja Storyboardia rakentamaan Objective-C-koodissa luodut näkymän elementit uudelleen visuaalisessa muodossa. Tämän jälkeen voidaan yhdistää näkymän haluttuun Swiftillä kirjoitettuun näkymän hallitsijaan. Riippuen projektista osa Objective-C-koodista voidaan myös kirjoittaa paljon lyhyemmin ja selvemmin Swiftille. Kuvassa 15 näkyy, miten ominaisuuden Swift-versiossa jaetaan näkymä storyboard-tiedostoon ja koodin toteutus näkymänhallitsijaan.

Kuva 15. Vanha sovellus käytti Objective-C-mallia, kun uudempi sovellus käyttää Swift-mallia.

Kääntämisprojektia aloittaessa on tärkeää varmistaa, että tulevat koodin muutokset eivät missään tapauksessa vahingoita App Storessa olevaa projektia, jonka takia luodaan versiohallintaan väliaikainen Git branch, jossa voidaan työstää projektia rauhassa. Uuden branchin luonti on hyödyllistä, koska voidaan tarvittaessa tehdä vanhaan Objective-C-versioon korjauksia tai päivityksiä. Tämä joustavuus voi olla hyvin tärkeää, jos kääntämisprojekti kestää pidempään ja samalla luodaan järjestelmällisyyttä versiohallintaan. Ennen kääntämisprojektin aloittamista on myös tärkeää tutkia mahdollisia apuvälineitä kääntämisen avustuksessa. Tässä projektissa käytetään Swiftify-nimisen Xcoden pluginin ilmaisversiota, joka avustaa Objective-C-ohjelmointikielen muuntamisessa Swiftille suorittaen päivässä rajoitetun määrän kääntämistä kehittäjän puolesta. Ilmaisessa versiossa koodia voidaan kääntää 2 KB ja maksullisissa versioissa kuukausi rajana on 1-100 MB riippuen versiosta. Maksullinen versio pystyy kääntämään isompia määriä päivässä. [21.]

Monissa projekteissa voi olla vanhaa Objective-C-ohjelmointikieltä ja uudempaa Swift-ohjelmointikieltä sekaisin, jonka takia on tärkeää, että tämä mahdollisuus otetaan huomioon kääntämisprojektissa. Jotta Swift- ja Objective-C-ohjelmointikielet voivat toimia

yhdessä, on tärkeää tehdä tarvittavat muokkaukset Objective-C-projektiin. Projektille on annettava Bridging-Header.h-tiedosto, johon kirjataan kaikki Objective-C .h-tiedostot, joita Swift-tiedostot voivat käyttää hyödyksi. Objective-C saa myös vastaavanlaisen tiedoston nimeltään ”projektinimi-Swift.h”, joka antaa Objective-C-koodin löytää ja käsitellä Swift-tiedostojen dataa.

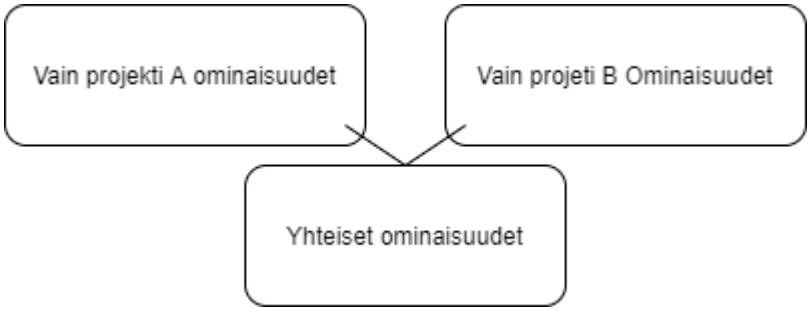
Osa käännettävistä ominaisuuksista voi olla myös hyvin vanhoja tai erilaisia yhdistettävään projektiin verrattuna, jolloin on suositeltavaa harkita ominaisuuden kokonaan uudelleen rakentamista Swift-projektiin. Tietyissä tilanteissa projektit voivat olla liitoksissa toisiinsa vain muutaman osuuden kanssa tai käännettävälle projektille kuuluva ominaisuus ja sen riippuvaisuudet eivät ole liitoksissa yhteiseen koodiin. Näissä tilanteissa on hyvä harkita ominaisuuden kokonaan rakentamista uudelleen suoraan kohdeprojektiin.

Ominaisuuksien kääntämisen jälkeen alkaa yhdistämisvaihe, jossa otetaan käännetyt kappaleet ja aloitetaan niiden siirto uudempaan projektiin. Ennen näkymien ja luokkien tuontia uuteen projektiin on suositeltavaa luoda kyseiseen projektiin toinen build target. Xcoden build target ja siihen liittyvät muutokset erottavat nämä kaksi projektia toisistaan. Targetin tarkemmat tiedot ja niihin liittyvät muutokset käytiin Xcode- ja Build Target -luvussa. Tämä vaihe voidaan jättää väliin, jos yhdistämisen jälkeen halutaan vain yksi sovellus käyttöön.

Targetin luonnin jälkeen projektiin voidaan alkaa kopioida käännettyä koodia. Uusille Swift-tiedostoille voidaan määrittää Target Membership -muuttuja kuvan 11 sivulla 17 tapaan. Tämän ansiosta vain projekti B target pystyy näkemään ja käsittelemään uusia Swift-tiedostoja. Tämä luo järjestelmällisyyttä projektiin ja erottaa alkuperäiseen sovellukseen liittyvät tiedostot päivitetystä sovelluksesta. Tietyissä tapauksissa kuitenkin molempien sovellusten tulee päästä käsiksi samoihin tiedostoihin, jolloin on mahdollista antaa molempien targetin käsitellä tiedostoa. Jos käytössä on ominaisuuksia, joiden halutaan olevan vain pääsyssä tietylle sovellukselle, mutta ominaisuudet löytyvät yhteisestä tiedostosta, voidaan jakaa tiedoston koodi vielä targetien kesken.

Joissain tilanteissa tiedostoa tarvitsee molemmat targetit, jolloin target-kohtaiset muutokset pitää tehdä koodin sisällä. Esimerkkikoodin 4 sivulla 17 esittämällä tavalla voidaan tehdä muokkauksia tai suorittaa sovelluskohtaisia toteutuksia huolehtimatta siitä, miten

muokattu koodi voisi vaikuttaa toiseen sovellukseen. Tämä on hyvin tärkeä osa sovelluksien jakoa, koska näin sovellukset voidaan jakaa toisistaan siististi. Lopputuloksena ominaisuudet jakautuvat kolmeen osaan: vain projektin A ominaisuudet, vain projektin B ominaisuudet ja yhteiset ominaisuudet (kuva 16).



Kuva 16. Yksinkertainen kuva esittää, mihin osiin sovellukset jaetaan.

Kuvassa 16 esitetään, miten koko projekti jaetaan kahden sovelluksen kesken. Vain projektin A ominaisuuksia ovat ominaisuudet, joita vain projekti A pystyy käsittelemään. Vain projektin B ominaisuuksia ovat ne ominaisuudet, joita vain projekti B pystyy käsittelemään. Loput ominaisuudet, joita molemmat sovellukset tarvitsevat kuuluvat yhteiset ominaisuudet -osaan. Hyödyntämällä Sovellusten erottelu -luvussa käsiteltyjä esimerkkejä sovellukset voidaan jakaa kahteen samoja osia hyödyntäviin sovelluksiin.

Näkymien tuonti vanhasta projektista uuteen on suhteellisen helppoa. Jos käännetyssä projektissa on storyboard-tiedostoja ja niihin luotuja näkymiä, voidaan ne avata lähdekoodina. Koska storyboard-tiedostot ovat XML-koodia, voidaan hakea halutut näkymät koodina ja kopioida ne haluttuun storyboard-tiedostoon uudessa projektissa. Siirtäessä näkymiä on huomioitavaa, että jos uudessa projektissa ei ole näkymälle asetettua näkymän hallitsijaa, pitää näkymän ja näkymän hallitsijan yhteys luoda uusiksi.

4 Toteutus

Teoriaosion jälkeen siirrytään käsittelemään käännösprojektin toteutusta. Osion aikana käydään kääntämisen vaiheet tarkemmin läpi ja esitetään hyötyjä ohjelmistokielen päivityksestä ja projektien yhdistäminen. Projekti alkoi alustavilla toimenpiteillä, joiden

tarkoituksena on mahdollistaa koodin kääntäminen ja varmistaa alkuperäisten versioiden turvaaminen.

4.1 Alustavat muutokset

Ennen kuin voidaan aloittaa mitään Swift-ohjelmointikielelle kääntämistä, meidän tulee tehdä alustavat muutokset projektitiedostoon. Ennen kuin pystytään käsittelemään Objective-C-tiedostoja Swift-luokissa, projekti tarvitsee Objective-C-bridging-header-tiedoston. Tämä tiedosto on oleellinen osuus, jotta Swift-luokat voivat keskustella Objective-C-koodin kanssa. Osa Swiftille käännettävistä osista hyödyntää Objective-C-pohjaista koodia, joka löytyy myös yhdistettävästä projektista. Näitä Objective-C-ohjelmointikielisiä koodeja ei ole käännetty Swiftille, koska ne ovat niin yksinkertaisia tai niille ei ole suunniteltu mitään muutoksia. Seuraavaksi muokataan projektitiedostoa ja luodaan -Swift.h-tiedosto, joka antaa Objective-C-tiedostojen käsitellä Swift-luokkia. Projektin ”Build Settings”-valikossa Packaging-osiossa ”Defines Module” pitää asettaa ”Yes”. Toisin kuin Objective-C bridging header.h, joka tarvitsee ”#import Objective-C.h”-tiedostot listattuna, ”-Swift.h”-tiedosto ei tarvitse mitään viittauksia eri Swift-luokkiin. Objective-C-koodi voi käsitellä Swift-tiedostoja, jos Objective-C tiedoston alussa kirjoitetaan ”#import ”Projekti nimi-Swift.h”” (esimerkkikoodi 5).

```
#import Foundation
#import UIKit
#import ”Projektin_Nimi-Swift.h”
#import “Objective-Ctiedosto.h”
```

Esimerkkikoodi 5. Objective-C-tiedostolle annetaan pääsy Swift-tiedostoihin projektin_nimi-Swift.h-tiedoston avulla.

Esimerkkikoodissa 5 näkyy Objective-C- ja Swift-tiedostojen ero, koska Swift-tiedostot eivät tarvitse erikseen kutsua #import ”Projektin Nimi-Swift.h” saadakseen yhteyden Objective-C-tiedostoihin. Vaikka -Swift.h-tiedosto ei tarvitse erillisiä viittauksia Swift-luokkiin, voivat Swift-luokat itse tarvita @objc-etuliitteen. Tämä etuliite mahdollistaa Objective-C-tiedoston lukemaan korkean tason Swift-luokkia. Esimerkkinä on Swift-luokan, -metodin tai -muuttujan käyttö. Näiden pohjalta Swift- ja Objective-C-koodit voivat kommunikoida keskenään, kääntämisprojekti voi alkaa.

4.2 Koodin kääntämisprosessi

Koodin kääntämisprosessi alkaa tiedostojen analysoimisesta. Tarkoituksena on selvittää, mitä tiedoston sisäinen koodi tekee ja mitä ulkoisia tiedostoja se tarvitsee toimiakseen. Järjestelmällisyyttä avustamaan luodaan lista sovelluksen käännettävistä ominaisuuksista ja niistä vastuussa olevista tiedostoista. Prosessin tuloksena voidaan tutkia, mitä ominaisuuksia tulee kääntää Swiftille ja mitä ominaisuuksia löytyy jo toisesta projektista. Esimerkiksi projekti B ja projekti A molemmat sisältävät Objective-C-ohjelmointikielellä luodun ominaisuuden, jonka tehtävä on esittää sille annettua materiaalia käyttäjälle. Ainoa ero näiden kahden välillä on materiaalin formaatti, projekti A esittää kuvia, kun projekti B esittää HTML-tiedostoja. Projekti B ominaisuutta on turha luoda uudelleen, koska ainoa ero projektien välillä tässä tapauksessa on kahden metodin toiminnallisuus ja se voidaan ratkaista esimerkkikoodin 4 esittämällä tavalla. Riippuen kuinka moneen osaan ominaisuus on alkuperäisessä ratkaisussa pilkottu, tulee lopputuloksena eri määrä uusia tiedostoja. Moni käännettävä ominaisuus tarvitsee ainakin kaksi uutta osaa toimiakseen uudessa Swift-projektissa. Muutamaa poikkeusta lukuun ottamatta kaikilla ominaisuuksilla tulee olemaan oma näkymä ja sitä hallinnoiva näkymän hallitsija.

4.2.1 Ominaisuuksien määrittely

Kaikkien ominaisuuksien määrittely on tärkeä osa kääntämisprosessia, sillä tämä avustaa järjestämään käännettävien osien kokonaisuutta. Määrittelyssä otetaan ominaisuus ja katsotaan, kuka kutsuu ominaisuutta, mitä näkymiä ominaisuus käyttää, miten ominaisuus toimii ja mitä muita tiedostoja ominaisuus tarvitsee toimiakseen. Esimerkiksi kolmella eri ominaisuudella projektissa oli yhteinen DetailViewController-luokka, joka oli vastuussa ominaisuuksien näkymien kutsumisesta käyttäjälle, tarvittavien tietojen siirtämisestä ominaisuuksille ja ominaisuuksien käytön toimesta muuttuvien arvojen esittämisestä käyttäjälle. Ominaisuudet tarvitsevat tietoja DetailViewControllerilta omissa toteutuksissa. Luokalle löytyi Swift-projektista vastine, jota voidaan käyttää kääntämisprosessin rajauksessa. Vastineluokkaa voidaan helposti muokata myöhemmin yhdistämisprosessissa vastaanottamaan käännetyt ominaisuudet. Kyseinen vertailu on hyvin tärkeää yhdistämisprosessin ja kääntämisprosessin kannalta, koska molempiin versioihin on voitu luoda muokkauksia. Nämä muokkaukset voivat aiheuttavat ongelmia koodin

toteutuksessa. Vertailussa huomattiin, että aikaisemmin mainitut kolme eri ominaisuutta oli jossain vaiheissa yhteydessä näihin neljään luokaan:

- DetailPagingController.m/h
- DetailViewController.m/h
- Metrics.m/h
- InfoView.m/h.

Tutkimalla uudempaa projektia DetailPagingController ja DetailViewController havaittiin toiminnoilta vastaavan erinimisiä luokkia: MDetailPagingController ja MDetailViewController. Näissä tilanteissa on tärkeää tutkia, mitä tietoja nämä kyseiset luokat antavat eri ominaisuuksille. Pääasiassa havaittiin näiden luokkien tarjoavan käännettäville luokille oleellisia tietokannasta haettuja arvoja, jotka ovat kriittisiä ominaisuuksien toiminnallisuudelle. Lisätutkimuksen tuloksena todettiin, että uudemmassa projektissa voidaan tuoda halutut arvot ominaisuuksille ilman isoja ongelmia. Toisaalta Metrics- ja InfoView-luokat olivat Objective-C-kielellä kirjoitettuja luokkia, joita käytetään edelleenkin Swift-projektissa. Objective-C-ohjelmointikielisten luokkien käyttö Swift-ohjelmointikielisissä projekteissa ei ole suoranaisesti ongelma Applen luomien toteutuksien ansiosta. Vanhat luokat voivat aiheuttaa kuitenkin ongelmia, jos Swift-koodissa ei oteta huomioon vanhan luokan kanssa työskentelyä tai vanhat luokat tarvitsevat päivityksiä toimiakseen suunnitellulla tavalla. Metrics-luokka pääasiassa tarjoaa nimensä mukaisesti mittoja, joita käytetään vakion omaisina arvoina ympäri sovellusta. Suosituin ominaisuus, jonka tämä luokka tarjoaa, on näytön mittoihin perustuva tieto siitä, mikä kännykkämalli käyttäjällä on (esimerkkikoodi 6).

```
if Metrics.isIphoneXR {
     //suoritettava koodi
}
```

Esimerkkikoodi 6. Metrics.m-tiedostoa hyödynnetään puhelinmallin selvityksessä.

Tämä (esimerkkikoodi 6) yksinkertainen toteutus palauttaa true-arvon, jos käyttäjän näytön mitat vastaavat IPhone XR -mallin mittoja. Kyseistä ominaisuutta käytetään esimerkiksi näkymien hienosäätöön, jossa tietty näkymän elementti ei esiinny halutulla tavalla ja tarvitsee muokattuja arvoja saadakseen halutun lopputuloksen käyttäjälle. InfoView ei eroa paljoa Metrics-luokasta, mutta tarkoitukseltaan tämä on hyvin erilainen. InfoView tarjoaa yksinkertaisia valmiiksi muokattuja hälytysviestejä, joiden sisällöstä vastaa hälytystä kutsuva luokka. Lopputuloksena on tyyliltään muokattu hälytys, joka on ulkonäöltään samanlainen sovelluksen kanssa, mutta jonka hälytyksen sisältö voi muuttua joustavasti riippuen toteutuksen tarpeesta. Jokainen näistä luokista on oleellinen osa ominaisuuksien esittämiseen ja toiminnallisuuteen, mutta ylimääräistä työtä voidaan välttää tarkalla tutkimisella.

4.2.2 Visuaaliset elementit

Visuaaliset elementit ovat tärkeä osuus jokaisen ominaisuuden päivityksen yhteydessä. Ominaisuutta kääntäessä on suositeltavaa harkita, mitä osuuksia voidaan luoda storyboard-tiedostoon ja voidaanko tiettyjen osuuksien toiminnallisuutta selventää. Esimerkiksi vanhassa Objective-C-versiossa yksi ominaisuuksista hyödyntää CaugeView-nimistä tiedostoa, joka on vastuussa tietyn mittarin esittämisestä käyttäjälle ja sen siirtämisestä, kun käyttäjä siirtää sormeaan näytöllä. Tämä mittari taittuu puoliympyrän muotoisessa arkissa ja käyttäjän sormea seuraa nuoli. Saman tuloksen pystyi luomaan hyödyntämällä UIKit-ohjelmistokehyksen tarjoamaa UIPanGestureRecognizer-luokkaa. Kuvassa 17 GaugeView-luokan mittari luotiin storyboard-tiedostoon, jota UIPanGestureRecognizer myöhemmin ohjaa.

Kuva 17. Näkymän nuoli annetaan näkymän hallitsijalle kuvan 5 tapaan sivulla 7. Tämän jälkeen UIPanGestureRecognizer voi manipuloida nuolta osoittamaan sormen sijaintiin.

UIPanGestureRecognizer on nimensä mukaisesti vastuussa tunnistamaan käyttäjän antamia vetäviä liikkeitä. Interface builderissa UIPanGestureRecognizer pystytään linkittämään kuvassa näkyvään nuoleen. Elementin hyödyntäminen vaatii sen yhdistämisen näkymän hallitsijaan. Hallitsijan sisällä elementille annetaan funktioita, joita kutsutaan, kun käyttäjä suorittaa ennalta määritetyn toiminnan käynnistäen funktion (esimerkkikoodi 7).

```
@IBOutlet var panGesture: UIPanGestureRecognizer!
@IBAction func handlePan(_ sender: Any) {
     if panGesture.state == .began || panGesture.state == .changed {
          let pos: CGPoint = panGesture.location(in: self.view)
          updateRotationToPosition(position: pos)
          needle.transform = CGAffineTransform(rotationAngle: self.rotation!)
     }
}
```

Esimerkkikoodi 7. handlePan-funktio suoriutuu, kun käyttäjä siirtää sormea näytöllä.

Esimerkkikoodin 7 panGesture on viittaus näkymän elementtiin ja tarvitaan itse handlePan-funktion toteutuksessa. handlePan käynnistetään, kun käyttäjä on siirtänyt sormeaan näytöllä. updateRotationToPosition-funktiossa lasketaan panGesturen sijainnin perusteella kulma, jonka mukaan nuoli voidaan siirtää osoittamaan sormen sijaintia puoliympyrässä. Näkymään asetettujen kuvien ja UIKitin UIPanGestureRecognizerin avulla kokonainen tiedosto pystyttiin tiivistämään näkymän hallitsijan sisälle. Vaikka koodin sisällä tehdään muokkauksia näkymään, kaikki näkymän elementit ovat esillä storyboardissa. Kokonaisen tiedoston muuttaminen yksinkertaiseksi funktioksi näkymän hallitsijaan luo oleellista järjestelmällisyyttä ja selkeyttää ominaisuuden rakennetta.

Kaikki käännettävät ominaisuudet hyödyntävät jotain näkymään vaikuttavia tekijöitä, tämän seurauksena Objective-C-koodista suurin osa voi olla pelkästään elementtien luomista, niiden sijantien määrittelyä ja toimintojen määrittelemistä (esimerkkikoodi 8).

```
self.gauge = [GaugeView new];
self.gauge.delegate = self;
self.gauge.translatesAutoresizingMaskIntoConstraints = NO;
[self.slideView addSubview:self.gauge];
...
```

```
[self.slideView addConstraint:[NSLayoutConstraint con-
straintWithItem:self.gauge attribute:NSLayoutAttributeCenterX relatedBy:NSLay-
outRelationEqual toItem:self.slideView attribute:NSLayoutAttributeCenterX mul-
tiplier:1.0 constant:0.0]];
[self.slideView addConstraint:[NSLayoutConstraint con-
straintWithItem:self.gauge attribute:NSLayoutAttributeCenterY relatedBy:NSLay-
outRelationEqual toItem:self.slideView attribute:NSLayoutAttributeCenterY mul-
tiplier:1.0 constant:-40.0]];
...
self.gauge.minValue = kEnergyLevelMinValue;
self.gauge.maxValue = kEnergyLevelMaxValue;
self.gauge.value = value;
```

Esimerkkikoodi 8. Lyhennetty esimerkki Objective-C-näkymän rakentamisesta.

Esimerkkikoodissa 8 näkyy, miten Objective-C-koodilla näkymät piti luoda koodin sisällä paisuttaen koko tiedoston kokoa. Jokainen elementti piti alustaa, muokata ja asettaa rajoitteita. Esimerkkikoodista 8 on poistettu toistuvia osuuksia, jottei koodia olisi liikaa. On tärkeää huomioida, että GaugeView sisältää vielä omat näkymän elementit ja niiden määritelmät. Uudessa Swift-koodissa hyödynnetään Interface Builderia luomaan kappaleet, niiden sijainnit ja järjestämään kappaleisiin liitoksissa olevat funktiot. Näkymästä voidaan myös helposti tuoda jokainen elementti koodiin, jos niitä halutaan käsitellä tarkemmin. Näkymän hallitsija voi muokata jokaista elementtiä koodin puolella ennen käyttäjälle esittämistä tai sen jälkeen. Yksinkertaisia esimerkkejä näkymän muokkauksesta ennen käyttäjälle esittämisestä ovat lokalisaatiot, jossa esimerkiksi UIButton- tai UILabel-tekstiosien sisältöä muokataan kännykän kielen mukaan (esimerkkikoodi 9).

```
contentHolderView.layer.cornerRadius = 9
tipBtn.layer.cornerRadius = 9
tipBtn.setTitle(NSLocalizedString("TIP_TITLE", comment: ""), for: .normal)
smallLabel.text = NSLocalizedString("ENERGY", comment: "")
smallLabel.adjustsFontSizeToFitWidth = true
```

Esimerkkikoodi 9. ViewDidLoad-osiossa suoritetaan yksinkertaisia muokkauksia näkymään.

Tässä esimerkissä näkymässä olevien elementtien kulmia pyöristetään, UIButton- ja UILabel-tekstikentät saavat lokalisaatiot ja UILabel-tekstikentästä tehdään joustava. Lokalisaatiot yleensä tehdään aikaisessa vaiheessa näkymän elinkaarta. Ensimmäinen muokkauksia esittävä vaihe on ViewDidLoad, jota kutsutaan, kun näkymä ja sen elementit on ladattu ensimmäistä kertaa. ViewDidLoad-funktioon yleensä kuuluu lokalisaatiot, elementtien tyylittely ja näkymän toiminnallisuuden kannalta oleellisten tietojen määrittely. Esimerkiksi UIPageViewController, jonka ViewDidLoad-osiossa määritellään kaikki UIViewControllerit, joita esitetään UIPageViewControllerin sisällä.

ViewDidLayoutSubviews on seuraava elinkaaren vaihe, jossa suurin osa näkymään halutuista kokoon liittyvistä muokkauksista tehdään. Näihin kuuluu esimerkiksi eri elementtien kokojen tai rajausten muokkaus kännykkämallin mukaan.

4.2.3 Objective-C- ja Swift-koodien yhteensopivuus

Objective-C- ja Swift-koodi pystyvät toimimaan yllättävän hyvin keskenään, jos Swift-koodille tehdään tarvittavat muokkaukset. Yksi ominaisuuksista löytyi muunneltuna Swift-projektista vanhana Objective-C-koodina, koska kyseiselle ominaisuudelle ei ole tehty mitään muutossuunnitelmia. Kuten aikaisemmin teoriavaiheessa mainittiin, koodissa voi olla osia, joiden kääntäminen Swiftille Objective-C-kielestä voi aiheuttaa enemmän ongelmia kuin hyötyjä. Toimeksiantajan kanssa katsottiin kyseisen ominaisuuden kääntämisen olevan turhaan resursseja kuluttava toimenpide, koska ominaisuudelle ei olla suunniteltu mitään muutoksia ja sen toimivuudelle tarvittavat osuudet eivät olleet kriittisesti liitoksissa muuhun koodiin.

Vanha Objective-C-kieltä käyttävä ominaisuus on yksinkertainen sivuihin perustuva näkymä. Näkymän tarkoituksena on esittää tiettyä ennalta määriteltyä sisältöä käyttäjille, kun he käyttävät sovellusta. Vanhassa versiossa sisältö muodostuu informatiivisista artikkeleista, joiden sisältö tulee eri HTML-tiedostoista. Uudessa versiossa sisältö muodostuu erilaisista kuvista, joita käyttäjät voivat selata. Ero versioiden välillä on käytännössä käyttäjille esitettävä sisältö (esimerkkikoodi 10).

```
- (void)loadContent
{
#if customFlagForTarget
//koodia
NSString *directory = [NSString stringWithFormat:@"/InfoHtml/%@/", [Localize
language]];
…
#else
NSMutableArray *images = [NSMutableArray arrayWithObjects:@"Theme1",
...
#endif
```

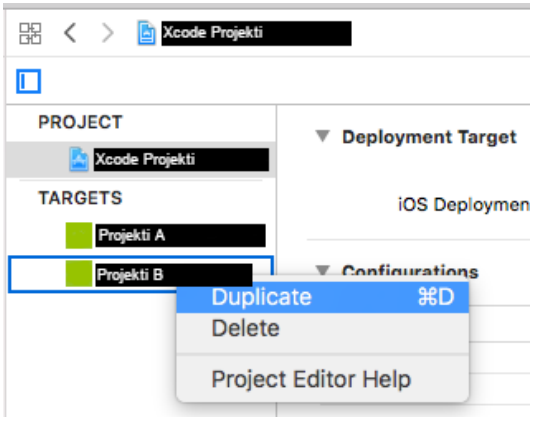
Esimerkkikoodi 10. customFlag-arvoja käytetään erottamaan sisältö.

Esimerkkikoodin 10 loadContent-metodissa suoritetaan riippuen sovelluksesta eri koodit. Jos käytössä olevalla sovelluksella on customFlagForTarget, etsitään InfoHtml-

kansiosta esitettävät html-sivut. Toisaalta jos sovelluksella ei ole kyseistä flagiä, haetaan lista kuvia. Kuten huomataan kyseisestä ominaisuudesta, Objective-C-koodin kääntäminen ei ole aina kannattavin asia. Vanha koodin toiminnallisuus voidaan yhdistää Swift-projektiin luomatta kokonaan uutta luokkaa. Moni vanha Objective-C-luokka on hyvin yksinkertainen ja toteuttaa pienen muusta koodista riippumattoman toiminnon. Vastaavanlainen ratkaisu on LoadingView.m ja LoadingView.h, jotka ovat vastuussa tietyn näköisen latausruudun esittämisestä ja piilottamisesta. Luokka on niin yksinkertainen ja joustava toteutuksessaan, että kaikki muut sovelluksen luokat voivat hyödyntää tätä pientä luokkaa omassa kokonaisuudessa. Toinen esimerkki hyödyllisestä Objective-C-luokasta on aikaisemmin mainittu Metrics.m ja Metrics.h, jotka tarjoavat monia yksinkertaisia lukuihin perustuvia toimintoja ja vakioita. Pääsyy, miksi kyseiset luokat voivat toimia ilman ongelmia Swift-koodin kanssa, on niiden yksinkertaisuus.

4.2.4 Sovellusten yhdistäminen

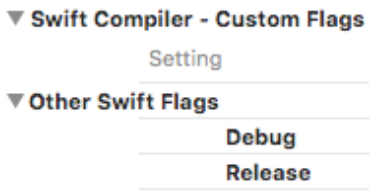
Projekti B on muutettu opinnäytetyön aikana Swift-koodia käyttäväksi sovellukseksi ja hyödyntää monia Xcoden ominaisuuksia. Projektit on tuotu pisteeseen, jossa sovellusten yhdistäminen onnistuu mahdollisimman pienellä vaivalla. Halutaan, että projektit hyödyntävät osittain samaa koodia, mutta silti pitävät ne erillään. Xcode tarjoaa build targetit, joiden avulla määritellään sovellus. Jokainen sovellus tarvitsee tämän tiedoston, sillä sen avulla App Store ja iOS tunnistavat sovelluksen. Yhdessä Xcode-projektissa ei kuitenkaan tarvitse olla vain yksi build target, vaan se voi sisältää useita targeteja (kuva 18).



Kuva 18. Target voidaan monistaa, koska ne hyödyntävät samoja osia.

Targetin monistaminen (kuva 18) luo projektiin uuden targetin lisäksi info.plist-tiedoston. Molemmat ovat tärkeitä targetin toimivuuden kannalta. Monistamisen jälkeen uuteen target-tiedostoon voidaan siirtää kaikki oleelliset tiedot vanhan projektin target-tiedostosta. Kyseisessä projektissa hyödynnetään pod-tiedostoa, johon pitää myös määritellä target-tiedoston riippuvuudet.

Targetit erotetaan toisistaan tiedostojen ja koodin kanssa. Tiedostoilla on oma target-muuttuja, joka kertoo, mitkä targetit pääsevät käsiksi tiedostoon. Muuttuja voidaan määritellä luonnin yhteydessä tai jälkikäteen tiedoston paneelista. Jos molemmat targetit tarvitsevat tiedostoa, voidaan tiedosto vielä tarkemmin jakaa koodissa Custom Flags-arvojen avulla. Custom Flags-arvot määritellään target-tiedon omaan Flags-muuttujaan (kuva 19).



Kuva 19. Build targetille voidaan antaa Custom Flags -arvoja, joita käytetään koodissa erottamaan targetien koodit toisistaan.

Uusi Custom Flags -arvo voidaan asentaa helposti painamalla kuvan 19 debug ja/tai release-kohtaa, jonka jälkeen syötteeseen kirjoitetaan Custom Flags -arvon nimi. Toinen hyötykäyttö Custom Flags -arvoille on debug- ja release-versioiden erottelu. Koodissa voidaan päättää, että osaa sovelluksesta ei käytetä, jos debug-versio on päällä tai toisteenpäin. Nämä voivat olla oleellisia kehitystyökaluja, joiden avulla luodaan esimerkiksi testidataa sovelluksen käyttöön. Esimerkkikoodi 10 esittää, miten Custom Flags -arvoja voidaan hyödyntää luokan sisällä erottamaan koodin toiminnallisuus targetin mukaan.

Targetin luonnin, määrittelyn ja Custom Flags -arvojen jälkeen voidaan yhdistää sovellusten tiedostot. Tiedostot ja näkymät voidaan siirtää yksinkertaisesti kopioimalla. Tiedostot voidaan halutessa ensin luoda uudessa projektissa ja tämän jälkeen kopioida sisältö vanhasta projektista uuteen. Näkymät voidaan siirtää samalla tavalla. Jos käytetään Xcoden ominaisuutta, voidaan nähdä storyboard XML-koodina sivun 4

esimerkkikoodin 1 mukaisesti. Koko näkymä voidaan kopioida ja siirtää vanhasta projektista uuteen. Kopioinnin takia näkymät pitävät kopioinnissa kaiken niihin liittyvän tiedon. Näihin tietoihin kuuluu myös näkymän hallitsijan nimi, joten varmistetaan, että näkymillä on oikeiden hallitsijoiden nimet.

Targetit voivat nyt toimia samassa projektissa ja uusia ominaisuuksia voidaan lisätä tarpeen mukaan. Projektiin lisättiin uusi localization.string-tiedosto, koska projekti A on lokalisoitu 5 eri kielelle ja projekti B vain kahdelle. Tiedostot erotettiin toisistaan Target Membership -muuttujan avulla ja lopputulos esiintyy kuvassa 12 sivulla 17. Tarvittaessa monia muita muokkauksia voidaan vielä tehdä molempiin sovelluksiin ja nykyisellä järjestelmällä voidaan välttää yhden sovelluksen sotkemasta toisen sovelluksen toiminnallisuutta.

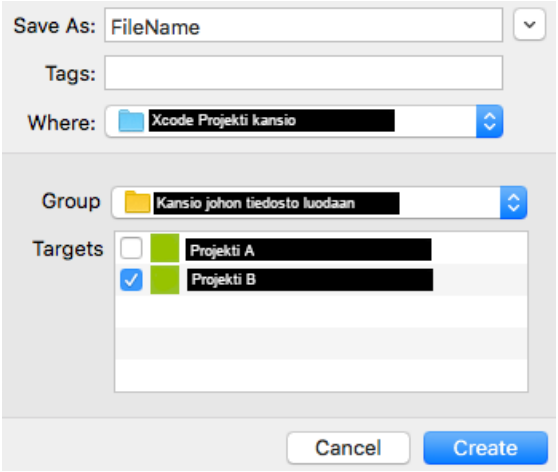
4.2.5 Uudet ominaisuudet

Uudet ominaisuudet ovat yksi tärkein syy, miksi koko yhdistämisprojektia aloitettiin tekemään. Ennen yhdistämistä uusien ominaisuuksien luonti oli työlästä. Ongelma kehityksessä oli se, että jos uusia ominaisuuksia haluttiin luoda, ne joudutaan usein luomaan kahdesti. Tämä johtui siitä, että projekti A käytti Swift-kieltä ja Xcoden ominaisuuksia, kun projekti B käytti Objective-C-kieltä ja hyvin vähän Xcoden ominaisuuksia. Uuden ominaisuuden kehitys vanhalle projekti B sovellukselle vaati kehittäjien tutustuttamista Objective-C-kieleen ja vanhoihin tapoihin rakentaa sovellusta. Tämä aiheuttaa turhaa työtä ja vaati usein saman työn toistoa ja muokkausta.

Huomattava esimerkki tästä on yksinkertainen ominaisuus, jonka tarkoituksena on esittää sovelluksen sisällä yrityksen verkkosivuja. Ominaisuus todettiin epäkäteväksi käyttäjille, mutta sitä ei haluttu poistaa kokonaan. Tämän takia ominaisuudelle tehtiin pieniä muutoksia. Ominaisuudelle annettiin uusi tarkoitus esittää informatiivista sisältöä käyttäjälle. Ominaisuuden monimuotoisuuden ansiosta sen sisältö muuttuu sovelluksen mukaan. Tarkoitus on esittää käännettävässä sovelluksessa hyödyllisiä linkkejä tai videomateriaalia yrityksen verkkosivulta suoraan sovellukseen. Materiaalin esittämistä haluttiin kuitenkin rajoittaa, jottei käyttäjille tulisi liikaa informaatiota kerralla. Yksinkertaiseksi ratkaisuksi luotiin sovelluksen sisälle menuvalikko, joka esittää viikkoihin jaettuja

videosarjoja. Kyseistä menuvalikkoa ja verkkosivujen esitystä hyödynnettiin myös uudemmassa sovelluksessa, jonne videoiden sijaan sisältö muodostui hyödyllisistä linkeistä.

Uusien ominaisuuksien kehitys nopeutuu huomattavasti yhdistämisprojektin ansiosta. Tulevaisuudessa molemmat sovellukset hyödyntävät samaa ohjelmointikieltä ja sovelluksen rakennetta, kuten aikaisemmissa luvuissa on käyty läpi. Uusia ominaisuuksia voidaan helposti luoda joko molemmille sovelluksille tai pelkästään toiselle. Sovellukset erotetaan toisistaan Build Targetin ja molemmille targeteille annettujen Custom Flags -arvojen avulla. Uudelle tiedostolle voidaan määritellä build target heti luonnin yhteydessä, kuten kuvassa 20 näkyy.



Kuva 20. Luotaessa uutta tiedostoa, voidaan valita, mitkä targetit pääsevät käsiksi tiedostoon.

Tiedoston luonnin (kuva 20) yhteydessä voidaan valita, mitkä targetit tietävät tiedoston olemassa olosta. Myöhemmin näitä voidaan muokata tiedoston Target Membership-muuttujan avulla. Xcode tarjoaa monia hyödyllisiä keinoja erotella sovellukset toisistaan, vaikka ne hyödyntävät osittain samaa koodia. Tätä ominaisuutta voidaan hyödyntää esimerkiksi projektin tyyppisessä tapauksessa tai testi-build targetin luomisessa.

4.3 Monetisaatiomalli

Sovelluksen nykyinen monetisaatiomalli on ilmainen ladata, mutta sovelluksen sisällä myydään eri tuotteita käyttäjille. Tämän mahdollistaa Apple, joka tarjoaa kehittäjille muutamia eri tapoja tuottaa rahaa sovelluksesta ja näihin kuuluu:

- ilmainen sovellus
- in-app-ostot
- suora osto.

Ilmainen sovellus nimensä mukaisesti on ilmainen ladata ja käyttää vapaasti, mutta yleensä sovelluksen sisälle asetetaan mainoksia, joiden esittämisestä kehittäjät saavat rahaa. Tämä voi esiintyä myös IAP eli In-App-ostoja sisältävän appin sisällä, jossa käyttäjät voivat ostaa itselleen lisäominaisuuksia esimerkiksi tarkempaa tietoa aikaisemmin käytetyn ominaisuuden arvoista tai sovelluksen sisällä olevat mainokset eivät enää näy. Yleensä tämän monetisaatiomallin tuotteina on myös tilauspaketteja, jotka jakautuvat kahteen eri ryhmään: uusiutuviin ja ei-uusiutuviin tuotteisiin. Uusiutuvat tuotteet veloittavat käyttäjää tietyn ajan kuluttua. Yleensä tämä on kuukausittainen maksu, mutta veloitusajat ovat joustavia. Ei uusiutuvat tuotteet ovat käytettävissä vain ennalta sovitun ajan ja veloittavat asiakasta vain kerran. Toisen tyyppisiä tuotteita ovat kertaostotuotteet, jotka myös jakautuvat kahteen: heti kulutettava tai pysyvä omistus.

Sovelluksen uusi monetisaatio-malli eroaa vanhasta tavasta, jolloin sovellus piti maksaa App Storessa kerralla ja käyttäjä sai sovelluksen pysyvästi itselleen. Monetisaatiomallin muutokseen oli kaksi syytä. Ensin käyttäjät pääsevät kokeilemaan sovelluksen ominaisuuksia ja päättämään, haluavatko he ostaa sovelluksen. Toiseksi tämä monetisaatiomalli on jo implementoitu uudempaan versioon mahdollistaen valmiin kokonaisuuden uudelleen muokkauksen yrityksen ja sovelluksen tarpeiden mukaan. Vaikka uudemmasta projektista voidaan lainata osaa koodista, on luotava myynti-ikkuna, järjestää tuotteiden osto, aktivoida ominaisuudet varmistamalla ostotapahtuman ja ominaisuuden avaaminen koodin puolella käyttäjille.

4.3.1 Myyntitapahtumat

Myyntitapahtumat alkavat myyntisivun esittämisestä käyttäjälle. Myyntisivun rakenne muodostuu muiden näkymien tapaan storyboardiin rakennetusta viewControllerista, johon liitetään näkymän hallitsija. Myyntisivu esittää kaikki tuotteet, joita myydään sovelluksen sisällä ja riippuen siitä, käyttääkö sovellus uusiutuvia maksuja vai ei, rakenteeseen tulee tärkeitä lisäyksiä. Jokainen myyntisivu tarvitsee yleistä tietoa tuotteista kuten nimi ja hinta sekä kuinka kauan tuote on käytettävissä. Samalla käyttäjille pitää olla pääsy sovelluksen yksityisyyskäytöntöihin ja käyttöehtoihin esimerkiksi linkin avulla. Uusiutuvien tuotteiden kanssa pitää ilmoittaa käyttäjälle Applen maksumenetelmät, joihin kuuluvat:

- Lasku tulee yhdistetylle iTunes tilille oston hyväksymisen jälkeen.
- Jos tilaus halutaan peruuttaa, se pitää tehdä ennen 24 tuntia nykyisen tilauksen loppumista.
- Tilaus uusiutuu 24 tuntia ennen nykyisen tilauksen loppumista.
- Kaikki ylimääräinen ilmainen kokeiluaika katoaa, kun käyttäjä ostaa tilauksen.

Jokainen muutos myyntisivun rakenteeseen pitää mennä Applen oman varmistustiimin läpi, minkä tarkoituksena on varmistaa, että sovellus noudattaa Applen antamia ohjeita. Jos sovellus ei noudata Applen ohjeita, niin se hylätään eikä päivitys mene App Storeen. Kun tuotteen myynti-ikkuna on hyväksytty ja käyttäjä ostaa tuotteen, sovelluksen pitää käsitellä tuotteen myyntiin liittyvät toimenpiteet.

Tuotteen myynti on suhteellisen yksinkertainen toimenpide. Myynnin käsittelyssä auttavat kaksi luokkaa: IAPHelper ja StoreObserver. IAPHelper on vastuussa suuresta osasta myynnin eteenpäin viemisestä ja sitä myös hyödynnetään kuittien varmistuksessa. StoreObserver on toisaalta vastuussa Applen kanssa kommunikoinnista. Myyntitapahtuma kuitenkin lähtee näkymänhallitsijasta, kun funktio esimerkkikoodissa 11 aktivoidaan.

```
@IBAction func purchaseFirstProduct(_ sender: Any) {
```

```
            let loadingView : LoadingView = LoadingView.
             sharedInstance()
            self.dismiss(animated: false, completion: nil)
            let product = IAPHelper.sharedInstance.
             productForProductId(productId:Constants.kCourse)
            if product != nil {
                IAPHelper.sharedInstance.purchaseProduct(product: product!)
                loadingView.show()
                IAPHelper.sharedInstance.transactionInProgress = true
            }
            loadingView.hide()
        }
    }
```

Esimerkkikoodi 11. Tuote ostetaan yksinkertaisessa funktiossa.

Esimerkkikoodin 11 funktio on IBAction eli Interface Builder -toimenpide, joka aktivoituu sender-elementin toimesta. Senderiksi voidaan määrittää esimerkiksi nappi. Funktion alussa etsitään LoadingView, joka on aikaisemmin mainittu luokka ja vastuussa yksinkertaisesta latausruudun esittämisestä. Tämän jälkeen IAPHelper hakee SKProduct-tuotteen tuotelistaltaan. Lista on aikaisemmin luotu, tuotteista muodostuva lista, joita sovelluksen sisällä myydään. Kun varmistetaan, että tuote ei ole nil, niin voidaan tarjota tuotetta asiakkaalle ja aloittaa myyntitoimet StoreObserver-luokassa.

```
let payment : SKPayment = SKPayment(product: product)
SKPaymentQueue.default().add(payment)
```

Esimerkkikoodi 12. purchaseProductin päätoimenpide.

Esimerkkikoodin 12 purchaseProduct funktiossa luodaan SKPayment, jonka tuotteeksi annetaan aikaisemmin määritelty tuote. Tämän jälkeen SKPayment annetaan SKPaymentQueuelle samaan tapaan, kun StoreObserver (esimerkkikoodi 2. s.12). SKPaymentQueue käyttää StoreObserveriä kommunikoimaan App Storen kanssa ja informoimaan muuta sovellusta myyntitapahtuman lopputuloksesta (esimerkkikoodi 13).

```
@objc
class StoreObserver : NSObject, SKPaymentTransactionObserver
{
    weak var delegate : StoreObserverDelegate?

    func paymentQueue(_ queue: SKPaymentQueue, updatedTransactions
     transactions: [SKPaymentTransaction]) {
        for transaction in transactions {
            switch transaction.transactionState {
            case .purchased:
                completeTransaction(transaction: transaction)
                break
```

```
            case .purchasing:
                break
            case .restored:
                restoreTransaction(transaction: transaction)
                break
            case .deferred:
                break
            case .failed:
                failedTransaction(transaction: transaction)
                break
            default:
                break
            }
        }
    }
```

Esimerkkikoodi 13. paymentQueue palauttaa lopputuloksen, joka lähetetään eteenpäin.

Esimerkkikoodissa 13 näkyy, miten StoreObserver on perinyt SKPaymentTransaction-Observerin, jonka avulla kommunikoidaan Applen App Storen kanssa. PaymentQueue palauttaa arvon perustuen switchille annetusta transaction-tilasta. completeTransaction- ja restoreTransaction-funktiot vievät IAPHelperille ostotapahtuman tuloksen. IAPHelper käsittelee sille annetun informaation ja aktivoi tuotteen tarjoamat ominaisuudet.

4.3.2 Tuotteen uudelleenaktivointi

Sovellukselle voi tulla tarve uudelleen tunnistaa käyttäjä ostotapahtuman jälkeen. Vastaavia tilanteita ovat esimerkiksi, jos käyttäjä vaihtaa puhelinta ja joutuu uudelleen asentamaan sovelluksen. Jos käyttäjällä on ostettuna tuote, tuotteen uudelleenaktivointi tapahtuu seuraavasti. Ensin käyttäjää kysytään kirjautumaan iTunes-tunnuksilla sisään, jotta voidaan hakea kyseisen käyttäjän App Store -kuitit. Applen antamat kuitit ovat vain kyseiselle sovellukselle kuuluvat tuotteet eikä mitään muuta (esimerkkikoodi 14).

```
let receiptURL: NSURL = Bundle.main.appStoreReceiptURL! as NSURL
do {
    let receipt: NSData = try NSData.init(contentsOf: receiptURL as URL)
    self.receiptData = receipt as Data
```

Esimerkkikoodi 14. Kuittidata haetaan Applelta.

Applen kuitit haetaan ja käsitellään ennen läpikäymistä. Kuitit käydään läpi JSON-objektina ja tarkoituksena on varmistaa, että kuitit ovat voimassa. Kuitti voi olla esimerkiksi

peruttu tai maksu epäonnistunut, jonka seurauksena kuittia ei tule käsitellä aktiivisena tuotteena. Kun hyväksytty kuitti löytyy, voidaan siirtyä ominaisuuksien aktivointiin.

Ominaisuuksien aktivointi perustuu pääasiassa IAPHelperin sisäiseen funktioon, jossa tutkitaan, onko ominaisuuden avaava tuote aktiivisena. Tilanteessa, jossa funktio palauttaa false-arvon, ominaisuudet eivät aktivoidu vaan käyttäjä viedään esimerkiksi myyntiikkunaan. Ominaisuus aktivoidaan, jos funktio palauttaa true-arvon. Tuotteiden vanhentuminen tarkistetaan aikaisemmin kirjattujen arvojen avulla ja muutamalla Applen funktiolla.

5 Yhteenveto

Opinnäytetyö alkoi ongelmallisesta tilanteesta, josta yritys kärsi kehittäessään kahta hyvin samanlaista sovellusta. Ajan kanssa toinen sovelluksista jäi taustalle ja siitä uupui oleellisia päivityksiä. Tilanne oli kehittynyt pisteeseen, jossa uusia ominaisuuksia oli todella hankala luoda molempiin sovelluksiin näiden eroavaisuuksien takia.

Opinnäytetyön tarkoituksena oli päivittää vanha sovellus ja yhdistää se aikaisemmin päivitetyn sovelluksen kanssa. Vanhan sovelluksen halutut ominaisuudet kirjattiin ylös ja aloitettiin ominaisuuksien uudelleen luonti. Ohjelmointikieli päivitettiin Objective-Cstä Swiftiin, kun se oli tarpeellista. Ominaisuuksien näkymät luotiin Xcoden työkalujen avulla vastaamaan aikaisemmin päivitettyä sovellusta. Molemmat sovellukset luotiin käyttämään osittain yhteistä koodia, mutta sovellukset pidetään eri sovelluksina Xcoden Build Target -ominaisuuden ansiosta.

Tuloksena on kaksi sovellusta, joita yritys voi jatkokehittää rinnakkain aikaisempaa pienemmällä resurssien kulutuksella. Tulevaisuudessa molemmat sovellukset voidaan jatkokehittää yrityksen haluamiin suuntiin ilman suurien eroavaisuuksien aiheuttamia ongelmia uusien ominaisuuksien kehityksessä.

Lähteet

1 Swift. 18.12.2019. Verkkoaineisto. <www.cleverism.com/skills-and-tools/swift/> Luettu 18.12.2019.

2 Apple. 18.12.2019. Verkkoaineisto. <developer.apple.com/swift/> Luettu 18.12.2019.

3 Macworld. 13.9.2010. OS X's ten most innovative features. Verkkoaineisto. <www.macworld.com/article/1154045/osx-innovative-features.html> Luettu 5.5.2020.

4 Xcode Releases. 18.12.2019. Verkkoaineisto. <xcodereleases.com/> Luettu 18.12.2019.

5 Amer, Ehab Yosry. 14.10.2019. iOS Storyboards: Getting Started. Verkkoaineisto. <www.raywenderlich.com/5055364-ios-storyboards-getting-started> Luettu 19.12.2019.

6 Apple. 27.10.2016. Verkkoaineisto. <developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/DesigningwithStoryboards.html> Luettu 19.12.2019.

7 Apple. 19.12.2019. Interface Builder Built-In. Verkkoaineisto. <developer.apple.com/xcode/interface-builder/> Luettu 19.12.2019.

8 Understanding Auto Layout. 21.3.2016. Verkkoaineisto. < https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/index.html > Luettu 28.12.2019.

9 Apple. 28.12.2019. Adaptivity and Layout. Verkkoaineisto. < developer.apple.com/design/human-interface-guidelines/ios/visual-design/adaptivity-and-layout/> Luettu 28.12.2019.

10 Cocoapods. 28.12.2019. Using CocoaPods. Verkkoaineisto. <guides.cocoapods.org/using/using-cocoapods > Luettu 28.12.2019.

11 Apple. 28.12.2019. Foundation. Verkkoaineisto. <developer.apple.com/documentation/foundation> Luettu 28.12.2019.

12 Apple. 6.1.2020. About App Development with UIKit. Verkkoaineisto. <developer.apple.com/documentation/uikit/about_app_development_with_uikit> Luettu 6.1.2020.

13 Apple. 6.1.2020. View Controllers. Verkkoaineisto. <developer.apple.com/documentation/uikit/view_controllers> Luettu 6.1.2020.

14 Apple. 6.1.2020. StoreKit. Verkkoaineisto. <developer.apple.com/documentation/storekit/> Luettu 6.1.2020.

15 Apple. 6.1.2020. In-App Purchase. Verkkoaineisto. <developer.apple.com/documentation/storekit/in-app_purchase> Luettu 6.1.2020.

16 Apple. 6.1.2020. Setting Up the Transaction Observer for the Payment Queue. Verkkoaineisto. <developer.apple.com/documentation/storekit/in-app_purchase/setting_up_the_transaction_observer_for_the_payment_queue> Luettu 6.1.2020.

17 Apple. 8.1.2020. Modeling Data. Verkkoaineisto. <developer.apple.com/documentation/coredata/modeling_data> Luettu 8.1.2020.

18 Apple. 8.1.2020. Core Graphics. Verkkoaineisto. <developer.apple.com/documentation/coregraphics> Luettu 8.1.2020.

19 Apple. 27.10.2016. Working with Targets. Verkkoaineisto. <developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/WorkingwithTargets.html> Luettu 13.1.2020.

20 Trapeznikov Eugene. 18.1.2016. How to Use Xcode Targets to Manage Development and Production Builds. Verkkoaineisto. <www.appcoda.com/using-xcode-targets/> Luettu 13.1.2020.

21 Petuschak, Alex. 20.12.2019. What is the maximum amount of code I can convert?. Verkkoaineisto <support.swiftify.com/hc/en-us/articles/360000110252-What-is-the-maximum-amount-of-code-I-can-convert-> Luettu 29.4.2020.